Lexical Analyzer

Igor Figueira Pinheiro da Silva - 15/0129921

University of Brasília

1 Motivation

The work presented here was developed for the Compilers course from the University of Brasília. All students must develop a compiler for a simplified version of the C programming language with some small changes. The development is divided into five steps: lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, and work presentation.

The new language that is going to be implemented is a simplified version of the C language. This will let us achieve our goal which is to have a functional compiler by the end of this course. The new language also contains two new data types. Type *set* is used to represent a new data structure that is similar to the mathematical set. Elements of a set are defined as a new data type as well, which is called *elem*. Both these new types have it's corresponding operations and value attributions.

2 Lexical Analysis

We introduce the lexical analyzer, which receives the source code as input and outputs a stream of all tokens and its identifiers. Errors are also checked and are identified accordingly. First of all, we define regular expressions to identify all the basic structures of the new language. Structures such as comments and whitespace characters must be ignored. Data types, operators, flow control keywords, and other important symbols have regular expressions specified to identify their type and extract the data correctly.

2.1 Data Structures and Functions

Column number and line number are saved in order to report unrecognized tokens. These values are changed using the *update_position* function. No data structures are used until this moment because currently the lexical analyzer only logs all the tokens that are recognized or returns an error. In the future, data structures such as the symbol table will need to be implemented.

There are some additional data structures that are not going to be used at the project's current state but will be needed later for the next steps. The symbol table, for example, has not been implemented yet but will be implemented using a list of lists or a C library that implements a hashmap data structure, in which the key's could be the symbols and the values the actual values associated to that symbol. The context-free grammar that will be used in the syntactic analysis can be found in the attachments section of this work report.

2.2 Tokens

Currently the tokens that are going to be sent to the syntactic analysis are not returned in the lexical analyzer for debug purposes. The tokens can be found in the "lex_analyzer.l" file in the format "<type,token>" that is used in all "printf" calls.

3 Testing

The files for testing the lexical analyzer can be found attached in zip file that was sent for this task.

3.1 Example 1

Example 1 is a correct example that contains a single variable declaration and some function calls and an if-else case.

3.2 Example 2

Example 2 is a correct example that contains some variable declarations and value attributions. It also contains a some operation inside a function call.

3.3 Example 3

Example is an incorrect example. The error occurs in line 5, in which "num1" is assigned the unknown symbol "#".

3.4 Example 4

Example 4 is an incorrect example and Example 4 is missing the closing double quote in line 2.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, & Tools. 2nd edn. Pearson, Boston (2007)
2. Levine, J.: Flex & Bison. 2nd edn. O'Reilly, California (2009)

Attachment 1 - Context Free Grammar

```
<program> ::= <declaration-list>

<declaration-list> ::= <declaration-list> <declaration>
                     | <declaration>

<declaration> ::= <func-declaration>
                | <var-declaration>

<type> ::= int
         | float
         | set
         | elem

<var-declaration> ::= <type> <id> ";"

<func-declaration> ::= <type> <id> "(" [<params-list>]? ")" <block-stmt>

<params-list> ::= <params-list> "," <param>
                | <param>

<param> ::= <type> <id>

<block-stmt> ::= "{" [<var-declaration>]* [<stmt>]* <return-stmt> "}"

<stmt> ::= <expression-stmt>
         | <if-stmt> ";"
         | <for-stmt> ";"
         | <func-call-stmt> ";"
         | <set-stmt> ";"

<expression-stmt> ::= <expression> ";"
                    | ";"

<expression> ::= <attribuition-expression>
               | <operation-expression>
               | <logical-expression>

<attribuition-expression> ::= <id> "=" <expression>

<operation-expression> ::= <add-expression>

<relational-expression> ::= <operation-expression> ["<="|"<"|">"|">="|"=="|"!="] <operation-expression>

<logical-expression> ::= "!" <relational-expression>
                       | <relational-expression> "||" <relational-expression>
                       | <relational-expression> "&&" <relational-expression>
                       | <relational-expression>

<if-stmt> ::= "if" "(" <expression> ")" [<stmt>|<block-stmt>]
            | <if-stmt> "else" <stmt>

<for-stmt> ::= for "(" [<attribuition-expression>]? ";" [<logical-expression>]? ";" [<operation-expression>]? ")"
```

```
<return-stmt> ::= "return" [<id>]? ";"

<add-expression> ::= <add-expression> ["+"|"-"] <mul-expression>
                   | <mul-expression>

<mul-expression> ::= <mul-expression> ["*"|"/"] <parenthesis-expression>
                   | <parenthesis-expression>
                   | <single-operator>

<parenthesis-expression> ::= "(" <add-expression> ")"

<single-operator> ::= <id>

<func-call-stmt> ::= <id> "(" <params-list> ")"

<id> ::= [<letter>|"_"]+ [<letter>|<digit>|"_"]*

<digit> ::= [0-9]

<letter> ::= [a-zA-Z]+
```